*Production Systems and Information Engineering*
Volume 8 (2019), pp. 69–87
https://doi.org/10.32968/psaie.2019.005



# C/C++ APPLICATIONS ON THE WEB

Martin Szabó
University of Miskolc, Hungary
Department of Information Engineering
sz.martin91@gmail.com

Károly Nehéz
University of Miskolc, Hungary
Department of Information Engineering
aitnehez@uni-miskolc.hu

[September 2017 and accepted November 2017]

**Abstract.** The JavaScript technology, especially in the latest five years, has been evolving very rapidly. In the world of WWW, of course, other technologies are also available for developers, but JavaScript is one of the best ways to develop applications for both traditional web and the latest mobile environments. The modern web application frameworks unfortunately do not support direct integration of C/C++ technologies. This problem can be solved by various utility software, e.g. the Emscripten compiler, which translates C/C++ codes into JavaScript leveraging LLVM technology as a transitional layer. Thus C/C++ programmers are not omitted from the world of Web and mobile development and they can reuse existing mature codebases, especially in the field of computer graphics. This paper describes theory, practical use and inherent potential deals of Emscripten technology. A native C++ OpenGL application will be used as a real-world example, demonstrating efficiency and flexibility of this technique. Performance evaluation of the JavaScript code compared to the native C/C++ application will also be presented. To promote better understanding, our source code is also available upon request.

*Keywords*: C/C++, JavaScript, LLVM, Emscripten, OpenGL

## 1. Introduction

Evolution of mobile and web technology development more and more excludes C/C++ programmers from the web client and mobile software market. The reason is that users prefer thin-client applications which can be launched on

mobile devices and in desktop browsers. Applications are written in JavaScript easier to be carried over between different browsers/platforms, easy to maintain and can be developed faster. These advantages can reduce the cost of development. Also, there are some different optimization techniques to increase performance, and reduce resource requirement of applications at the same time. In contrast, high-level C/C++ programming language is harder to port between different platforms, and require a recompilation in the most cases. But, in case of complex and resource-intensive tasks (i.e. game engine development), it's still worth choosing C/C++ technologies.

Based on the current studies [1] programmers prefer more C/C++, than JavaScript. In additional, a lot of complex systems have been written in C/C++. These applications are constantly under maintenance and upgrades, so they are still doing their job well (so they are still able to do their job well). So the question is, why cannot we use these great systems via web or mobile devices? The traditional idea is that, we should rewrite source code of the whole application. But this problem can be solved with a new approach, via Emscripten technology.

Using Emscripten, applications written in C/C++ programming language, can be translated to JavaScript. During compilation process, the code translated to LLVM bit code, so it can be optimized. The result will be a less resource-intensive, but outstanding performance JavaScript-based software. Unlike the original code, the applications have been created this way, will be able to run on mobile devices and in browsers. So the recompiling code automatically provides the benefits of JS technology.

This article can be divided into four parts. The first part contains description of the essential theoretical background of Emscripten framework, including a solid theoretical description of JavaScript, and its advantages and disadvantages will be presented. As an integral part of Emscripten, we present the structure and the most important functions of the LLVM compiler. The second part of the paper presents the Emscripten itself, and contain a guide required to use the technology. The third section shows how to use the framework in practice, via a simple graphical application. It includes the constraints of the technology, and other tasks required to achieve a proper final result. The last part describes our experiences gained through the testing of the framework, and contains some benchmark data, we got from the C/C++, and the translated JavaScript application.

## 2. Background

A basic knowledge of C/C++ language is required to use Emscripten technology. The usage of the compiler can be problematic without understand the code appears at the frontend of the framework (even if we have a lot of source files (even in the case of having a lot of source files). There will be more information about this issue in Section 4. Furthermore, the JavaScript code generated by the Emscripten does not require additional development work, it is necessary to briefly introduces the script language, including its advantages and disadvantages.

JavaScript [2] is an object-oriented, prototype-based scripting language that is mainly used for Web sites, either as an integral part of an html file or in a separated JS file. It started to spread in the 1990s, and still widely used as the part of the websites, with HTML, and CSS. Other programming languages also can be run on the web, such as Java, and Flash, but in these cases, the extensions are not integrated into the browser applications, but must be manually installed. Furthermore these are not platform independent tools, for example usually cannot be run on iOS devices. Because of these issues JavaScript becomes the most widely used programming language to create the client side of web applications. The advantages of the language are the following. software created by JavaScript can be run on desktop, and mobile browsers too. It runs on client side, the development is relatively easy, quick, and does not require recompilation. Disadvantages include the fact, that it is browser dependency and slow speed. Although it standardized, but different engines (layout engines) can interpret it in different ways, which leading to operational inconsistencies. Compared to C/C++ applications, it significantly provides less performance, but with optimization procedures of LLVM compiler (which is the part of the Emscripten technology) a majority of the lost performance can be recovered.

Before creating the JavaScript code, Emscripten compiles the C/C++ sources into LLVM bit code. This procedure can give a quasi-optimal code as a final result, which is already ready to run in browsers. LLVM compiler is a very important, and useful part of Emscripten, so this paper deals with its theoretical background as well.

We have collected and investigated other technologies designed to run a specific programming language on the Web. For example Google Web Toolkit [3] which translates Java to JavaScript, Pyjamas (Python → JavaScript), SCM2JS [4] (Scheme → JavaScript), and AFAX [5] (F# → JavaScript) frameworks are serve this purpose too. But, in this paper we are focusing C/C++, and these tools are not suitable for our goals, so we will not deal with them.

### 2.1. Low Level Virtual Machine (LLVM)

Emscripten framework uses LLVM to optimize the output of the JavaScript code. The LLVM compiler has been designed to optimize any programming languages in compile-time, link-time, run-time, and in idle time. A robust open source application, that competes with GCC in terms of compilation speed and the performance of the generated code. As a consequence, it has been widely used in both academia and industry [7].

The code representation describes the code as an abstract RISC-like (reduced) instruction set, but also provides higher level information for more efficient analysis. Including type information, explicit control flow graphs, and an explicit dataflow representations. It supports language independent instruction set and type systems. The instructions stored in SSA (Static Single Assignment) form, which helps to simplify analysis of dependencies between variables [8].

The LLVM code representation has the following novel properties [7]:

1. Low-level, language-independent type-system for implement data types and operations from high level languages, eliminating the primitives at all stages of optimization. The type system includes type information used by sophisticated (but language-independent) techniques, such as algorithms for pointer analysis, dependence analysis, and data transformation. In other words: simple, language-independent type system, that exposes the primitives generally used to implement high level programming language features.
2. Instruction set for performing type conversions and low-level address arithmetic without losing the type information.
3. Two, low-level exception-handling instructions for effectively implementing high-level language specific exceptions with simple methods.

The LLVM compiler is source-language-independent due to low-level instruction set, and memory model. The technology slightly offer more than the standard assembly languages, and the type system does not prevent to add type information to the code. It also does not have specific runtime requirements [7].

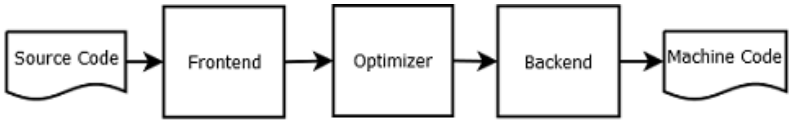


**Figure 1.** Simplified model of LLVM

The simplified usage of the compiler framework can be described as follows: The source code at the frontend (written in any programming language) translated to machine code. Then different optimization methods (depends on the user choice) optimize the source. Finally the result will be runnable code at the backend, written in an appropriate programming language for the target architecture (Figure 1).

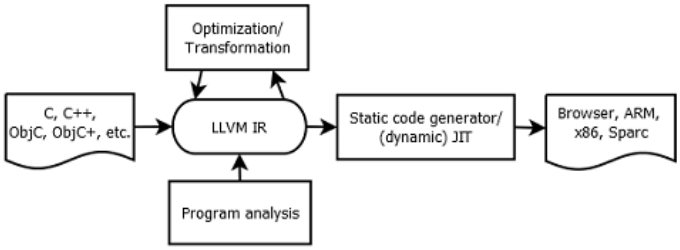


**Figure 2.** Detailed model of LLVM

The LLVM-based compiler allows to compile the high level programming language resources to LLVM IR language (Figure 2). The IR (Intermediate Representation) is assembly-like low-level programming language that allows optimization, code compilation, and static analysis. The resulting code runs on a variety of systems, including for example the X86/64, and the ARM architectures. The technology allows static translation (as usual in the case of the GCC), but also supports dynamic JIT (Just-In-Time) translation. The project created mainly for C/C++ front-end, but has the ability to compile any other programming language too, for example Haskell, Scheme, Scala, Objective C etc. [6]

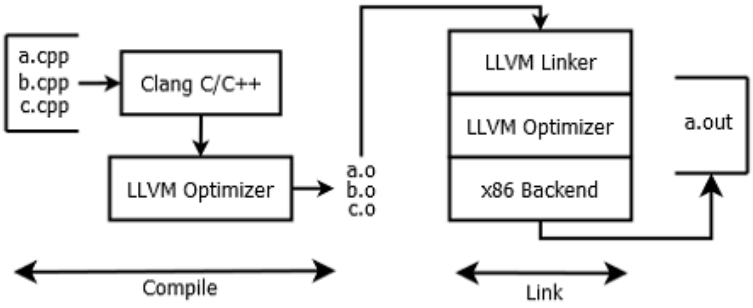


**Figure 3.** LLVM LTO

Most of the modern compiler environments support optimization in linking time (LTO)(Figure 3). The disadvantage of LTO is that, it can deal with only one unit simultaneously (like a C file, with its headers) and cannot optimize a larger context. LLVM compilers, (like Clang) support this type of optimization with the *–flto*, and *–o4* commands. These settings indicate to the compiler, that it needs to produce .o file from the LLVM byte code instead of a native object file, and that, the code generation have to be in the linking phase, as shown in Figure 3. The linker detects, that the generated .o files are not native object files, but contain LLVM byte code, therefor it reads every byte code file into the memory, and after linking executes optimization processes on the entire code. As the optimizer unit oversees the entire structure now, it can perform its tasks more effectively [9].

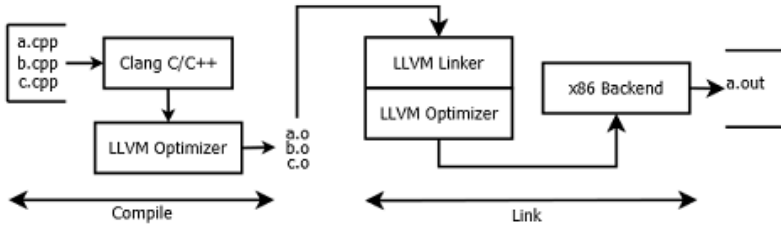


**Figure 4.** LLVM ITO

Installation time optimization (ITO) can take into account special features of the target architecture as well (eg. MMX, SSE, AVX). The advantage of this method is that, the optimizer unit will recognize the device specifications in all cases, so the architectural benefits are exploitable(Figure 4).

## 3. Emscripten Conception

Applications, written in C / C++ language are generally "make life easier for the computer" so it can achieve higher performance. In contrast, applications written in JavaScript make life easier for programmers, but some loss in performance occurs. If high performance is essential for a given kind of software (for example, video-game engines, distributed systems etc.) the C/C++ language is inevitable. Of course, it is not expected to make the same application for multiple platforms, by the reason of the short deadlines, generated by rapidly developing technology. This problem can be solved by the Emscripten technology that converts the C/C++ source code for the web and/or the mobile version of the same product, with minimal modification (adds Emscripten specific methods etc., see Chapter 4). Build-in LLVM compiler also eliminates

the above-mentioned performance loss using various optimization methods. The behavior of the resulting software is same as the original application, but its availability (due to the web environment) and its performance (due to the optimization) is greater (Figure 5).


**Figure 5.** Emscripten compiling conception [11]

Emscripten framework is written in JavaScript language, and it can be downloaded from the following link under MIT open source license: http://www.emscripen.org. The compiler is designed to fix parts of the original high-level structure of the code, that were lost during the compilation to low level LLVM byte code, when creating the JavaScript code. In the following we describe the LLVM – JavaScript compilation process, and present the Relooper algorithm, which is able to create high level loop structure from the low level branching data.

There are two available methods to use this technology: the first option is the pre-translation (static) mode, the second is the runtime translation (dynamic) mode. Then we translate the resulting code to JavaScript with the help of Emscripten. The latter option is useful when the run time of the language is written in a programming language, that has, but the language itself has no frontend. For example, currently Python does not have a frontend, but it is possible to translate CPython (Python standard C implementation) to JavaScript, and run Python code. Frontends for various languages exist, including C/C++, and also various new and emerging languages (e.g., Rust) [11].

### 3.1. Optimization process

Many (mostly big, but sometimes smaller) application's code, during the software development processes become too complex, so the errors almost inevitable, and the final performance can be far below what is expected. The most common problem is that the developers do not invest enough emphasis on optimization, and do not clarify various dependencies like dynamic libraries, etc., because these limitations have not been appeared on the platform, which the application is designed. If we would like to run the application on another hardware, it might not be able to ensure proper performance, and/or it will be difficult to port to other platform.

Emscripten itself does not provide optimization automatically without the appropriate command line settings. With default settings, the framework generates the JavaScript code without any optimization (i.e. it contains more than necessary variable declaration). For example it stores every variable in an array, and controls the flow of execution using a switch-in-a-loop, instead of normal JavaScript loops and ifs. Before compiling the C/C++ code to JavaScript, the Emscripten technology optimize it with the help of the built-in LLVM, furthermore the Closure Compiler give some more optimization for the .js code afterwards. These procedures remove unneeded variables, eliminate dead code and inline functions [11]. There are two important optimization methods, as follows:

1. **Variable nativization:** converts every variable that are on the stack to native JavaScript variables. The Emscripten tries to implement this process to the maximum number of variables during the optimization processes. Except those variables, which were used outside of a function, or were passed to another method.
2. **Relooping:** recreate the high level loop and of structures from the low level LLVM assembly code.

### 3.2. Relooper algorithm

The Relooper is the most complex module of Emscripten. It generates high-level JavaScript flow structures (loops, ifs), Emscripten code blocks from set of labeled fragments of code [11]. The JS engines are designed to run the codes as fast as possible, so the structure of the code have to be manufactured to comply with this restriction. To present the steps of the Relooper algorithm we have to define the three different blocks of Emscripten. These are shown below [11]:

1. **Simple:** contains one internal label, and a reference to the Next block, which the internal label branches to.
2. **Loop:** it represents a basic loop, and contains two internal sub-blocks (the Inner block, that appears inside the loop, and the Next block, that appear outside the loop).
3. **Multiple:** it represents branches joining each other, and also contains two internal sub-block, the Handled, and the Next blocks.

The steps of the Relooper algorithm are as follows [11]:

1. Receive a set of labels and entry point. The goal is to make a block from the labels.
2. Tracking one of the possible execution path, it calculates for every label which other labels it can reach.

3. If there is only one entry point, and cannot return back, then creates a *Simple block*, where the entry point is the internal label, and the *Next block* contain every other label. The entry points of the *Next block* are those points, which the internal label can branch.
4. If it can return all of the entry points, creates a *Loop block*, whose *Inner block* contains all of the labels, which can reach one of the entry points, and whose *Next block* contains every other labels.
5. If there are more than one entry point, then it creates a *Multiple block*. Find all labels that cannot be reached by other entry points for all entry point.
6. If it cannot create a *Multiple block*, then it prepares a *Loop block* as described above.

## 4. How to Use Emscripten

The installer available at the following link: http://www.emscripten.org.
A lot of tutorials are available on the same link too. Source codes can be found in the Emscripten's sub-folder after the installation. The framework is available on Windows, Linux, and iOS as a command-line environment. The official website provides detailed descriptions about the technology, but in this paper we present a more complex example via a C/C++ graphical application. The manual does not complete, is shows only the functions we utilized. The demo application is available at the following link: http://goo.gl/YmV2EA. (Figure 6)shows the the demo application.

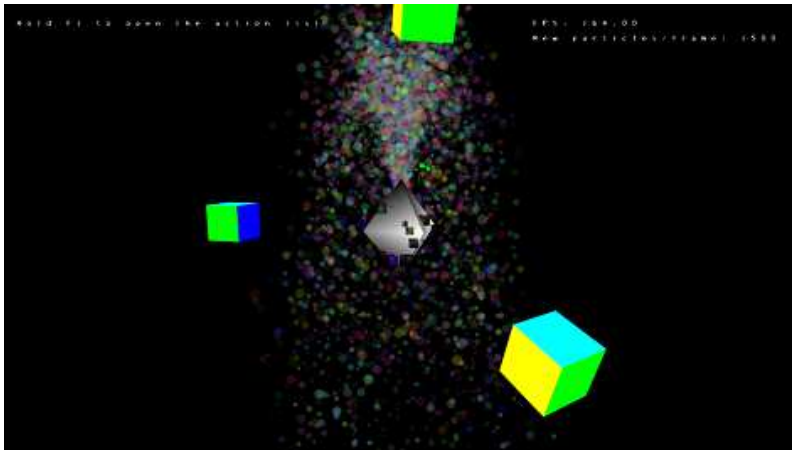

**Figure 6.** Demo application

### 4.1. Coding process

Usually if the source code does not contain graphical elements, then it does not require interventions before use. However, most of the code of graphical applications have to be prepared for different environments (i.e. if the code contains infinite loop [gameloop]). This is the job of application developers. The code transformation process will work, but the JavaScript code will not run properly in the browsers. The JavaScript code can run only in one thread, so these solutions cannot be used without modifying the code. To solve this program the developers of the Emscripten technology creates the *emscripten_set_main_loop (function, int fps, int simulate_infinite_loop)* method. The *function* parameter indicates the primary method have to be done by *while* loop; *fps* means frames per second (0: unlimited); and the last parameter represents the infinite loop simulation (0: run only once, 1: infinite loop). In our application we used this method as follows:

```
#ifdef EMSCRIPTEN
#include <emscripten.h>
#endif

void loop(){
/* draw models, translations, rotations */
/* draw particles */
}
int main( void ){
/* initializing GLEW */
/* Creating matrices, textures, particle system, models (coordinates, colors) */
#ifdef __EMSCRIPTEN__
  emscripten_set_main_loop(loop, 0, 1);
  #else
  do{
  loop();
  }
  while( glfwGetKey(window, GLFW_KEY_ESCAPE ) != GLFW_PRESS &&
```

**glfwWindowShouldClose**(window) == 0 );

**#endif**

**}**

In order to use the Emscripten specific methods in our code, we have to include *emscripten.h* header file. With the use of *#ifdef* , we can compile the source code and run the application as a C/C++ and Emscripten versions too without performing any changes in the code. So, if we would like to run it as a C/C++ application, Emscripten specific parts will not compile, on the other hand, if we would like to transform the source code to JavaScript, then these parts will compile too.

Developers of the Emscripen offer the SDL window manager library to use. In contrast we used GLFW3, and our experiences show that this library is fully compatible with the Emscripten as well. However, the necessary header files are not included in the framework. The solution to this problem is simple, we only need to copy the appropriate files into the Emscipten's *include* folder. But the SDL tools do not require any action from the developers.

To move and to rotate three-dimensional shapes, it is required to perform matrix transform operations. The GLM libraries contain the necessary methods for this. This header files are not part of the Emscripten framework, but easy to install them, as described above.

The application has been made in OpenGL 3.3 environment. To draw the models, we originally used GLSL 3.3 (OpenGL Shading Language) language, which is a high-level language based on the C programming language syntax. Most of the sample programs we found on the internet use this version as well. However, we realized that the Emscripten only supports the WebGL compatible GLSL ES 1.0, and 2.0 shader languages, and GLSL 3.3 code cannot be automatically converted to one of these. We can solve this problem with manual code modifications. The following table shows the differences between the two mentioned shader language implementation through the vertex and the fragment shader we used in our application.

Table 1: Differences between the vertex shaders

| GLSL 3.3 | GLSL ES 1.0 |
|---|---|
| `#version 330 core`<br>`layout(location = 0) in vec3`<br>`vertexPosition modelspace;`<br>`layout(location = 1) in vec3`<br>`vertexColor;`<br>`out vec3 fragmentColor;`<br>`uniform mat4 MVP;`<br>`void main() {`<br>`gl_Position = MVP *`<br>`vec4(vertexPosition_modelspace, 1);`<br>`fragmentColor = vertexColor;`<br>} | `attribute vec3`<br>`vertexPosition modelspace;`<br>`attribute vec3 vertexColor;`<br><br>`varying vec3 fragmentColor;`<br>`uniform mat4 MVP;`<br>`void main() {`<br>`gl_Position = MVP *`<br>`vec4(vertexPosition_modelspace, 1);`<br>`fragmentColor = vertexColor;`<br>} |

Table 2: Differences between the fragment shader

| GLSL 3.3 | GLSL ES 1.0 |
|---|---|
| `#version 330 core`<br>`in vec3 fragmentColor;`<br>`out vec3 color;`<br>`void main() {`<br>`color = fragmentColor;`<br><br>} | `precision mediump float;`<br>`varying vec3 fragmentColor;`<br>`/* varying vec3 color; */`<br>`void main() {`<br>`gl_FragColor =`<br>`vec4(fragmentColor, 1.0);`<br>} |

Table 3: Summary of the differences between the two versions

| **GLSL 3.3** | | **GLSL ES 1.0** | |
|---|---|---|---|
| **vertex shader** | **fragment shader** | **vertex shader** | **fragment shader** |
| #version xxx core | #version xxx core | – | precision mediump float |
| layout(location = x) in | in | attribute | varying |
| out | out | varying | varying |
| | out *variable's name* | | gl_FragColor |
| | texture | | texture2D |

There were no further issues has been reported during the translation of the sample application. However, we tried to compile a complex video-game engine, but a new issue appeared. The software contained some old, OpenGL 1.x specific methods that Emscripten was unable to translate correctly. This problem should be solved by the *–s LEGACY_GL_EMULATION = 1* argument, but based on our experiences, some functions are not useable yet. Note that: the framework could not translate the following OpenGL methods:

1. `glListBase`
2. `gluBuild2DMipmaps`
3. `glDeleteList`
4. `glDrawPixels`
5. `glLightf`
6. `glVertex2D`
7. `glGenLists`
8. `glEndList`
9. `glCallLists`
10. `glNewList`

### 4.2. Compiling process

The compiler framework can only work in a command-line environment. For large-scale projects, it is subservient to create a Makefile, because of the numerous source files and other auxiliary files, like models, shader, and textures. This prevents (using different technologies, changing name, or number of the source files etc.) you from typing the – usually long – commands into the command prompt again.

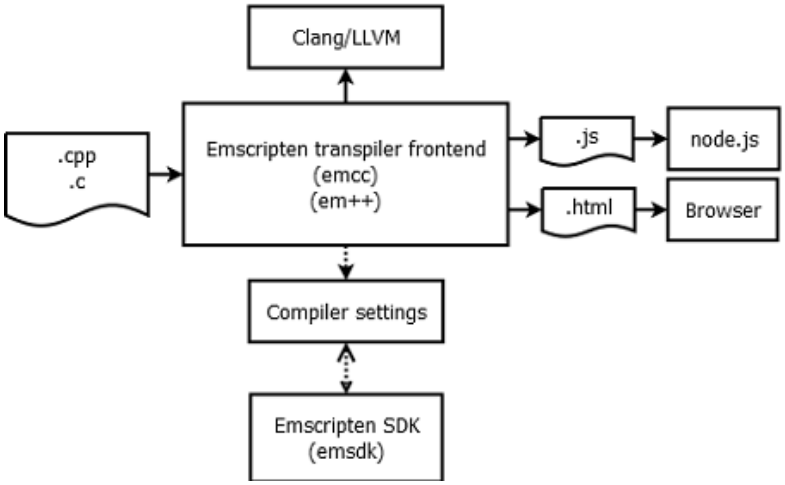


**Figure 7.** Detailed model of Emscripten

The compiler input is the C/C++ files and the output depends on the extensions specified (.js, .html). It can be .bc (or .o), .js, and .html. The first represents pure LLVM byte code, the second represents JS code, and the last one represents JS code embedded into html template. Tracking and setup the runtime parameters are easier using the html template. The code obtained in this manner will be immediately executable in desktop and mobile browsers (Figure 7). The LLVM byte code cannot run on its own.

The command which is required to compile the application is divided into five parts. The first part is the emcc (in case of C source), or the em++ (in case of C++ source) command. This is followed by the list of source files, then the switches required for the used technologies. If other files are needed to run the application properly (models, textures, shader etc.) Emscripten has to preload these, so we need to mark them as well. In this case, a .data file will be generated (besides .js and .html files), which will contain these information. The last part of the command will indicate the output file.

*emcc main.c shader.c –s USE_SDL = 2 –preload-file texture.jpg –o index.html*

Switches and other necessary commands we used in our application are shown in the following table with explanation:

Table 4: Command line switches for the different technologies [10]

| | |
|---|---|
| -s USE_SDL = 2 | allow SDL2 |
| -s USE_SDL_IMAGE = 2 | allow SDL_image |
| -s USE_GLFW = 3 | allow GLFW3 |
| -s FULL_ES2 = 1 | allow GLSL ES2 |
| -s LEGACY_GL_EMULATION=1 | force to handle OpenGL 1.x methods |
| -s DEMANGLE_SUPPORT = 1 | convert class and function names |
| -std=c++11 | handle c++11 specific opportunities (i.e. lambda) kezelése |
| –preload-file dir/fájl_neve | load files (without file name the whole directory will be loaded) |

Each switch performs different operations depending on what is required. (Table 5) We analyzed the results given by all the available switches. Our experiences will be summarized later.

Table 5: Switches for optimization [10]

| | |
|---|---|
| -O0 | No optimization |
| -O1 | Simple optimization. Fast, but not really increase the performance. Eliminates run-time exception handling assertions. Relooping. |
| -O2 | Same as the –O1 switch, but extended with JS optimization. It is useful in preparing the final version of the application. The translation is slow, however, this method provides the smallest and the fastest JS code. |
| -O3 | Same as the –O2 switch, but provides more JS code optimization. |
| -Os | Same as the –O2 switch, but provides ,,extra" optimization, that reduces the size of the code, and increases the speed of the application. Only for byte code optimization. |
| -Oz | Same as the –Os switch, but takes further code size reduction. Only for byte code optimization. |
| -s DISABLE_EXCEPTION _CATCHING = 0 | Enable exception handling |
| -s AGGRESSIVE_VARIABLE _ELIMINATION = 1 | Eliminate unnecessary variables |
| -s ALLOW_MEMORY _GROWTH = 1 | Unlimited memory usage |

As explained above, it is always useful to create a *makefile* for our projects – especially if it contains a large number of files – to facilitate the translation process, and the practical testing. With this method it is enough editing the *makefile*, instead of typing all of the commands again and again. Of course, the resulting *makefile* is not compatible with the Emscripten system, so we need to perform some changes. The converted, Emscripten compatible *makefile* contains the following code:Emscripten makefileW9067CC = emcc
SOURCES := $(wildcard *.cpp)
LDFLAGS = -O2 –llvm-opts 2
OUTPUT = glcore.html
F = –preload-file
all: $(SOURCES) $(OUTPUT)
$(OUTPUT): $(SOURCES)
$(CC) $(SOURCES) –bind -s USE_GLFW = 3 -s FULL_ES3 = 1 -std = c++11

```
$(F) ColorFragmentShader.fragmentshader $(F) TransformVertexShader.vertexshader
$(F) TextVertexShader.fragmentshader $(F) TextVertexShader.vertexshader
$(F) Particle.vertexshader $(F) Particle.fragmentshader $(F) particle.DDS $(F)
font.DDS $(LDFLAGS) -o $(OUTPUT)
clean:
rm $(OUTPUT)
rm $(OUTPUT).mapW9067
```

## 5. Performance Analysis

Based on our experiences there are some cases, when the use of this technology is simple, and the C/C++ source code does not require any code modification. However, there are – mainly – graphical applications, where we have to modify the source code of the original C/C++ application. Fortunately, these changes are quite simple, only infinite loops ([*gameloops*]) should be changed as described above.

Another problem is that, although a lot of files needed to develop a graphical software are built into the compiler, the service is not complete, some of the methods we have to use, are not part of the package. Mostly, these are the functions used in the older OpenGL 1.1 technology. The personal solution to this problem is not an easy task, it requires programming work within the Emscripten framework. According to our research, the most appropriate solution is the Regal graphics library package [14]. Furthermore there are some other tools, which are unavoidable in many cases, but the framework does not include them. For example the GLM library package for matrix operations, or the GLFW3 windows manager technology (instead of SDL/SDL2). However, in these cases, we only need to copy the necessary header files into to include folder of the Emscripten, as described above.

The files (models etc.), used by the framework, need to be preloaded, and this must be done by the developer. It is not a problem, but it may cause a minor inconvenience. However, there are many other suitable solutions, for example setting the root folder, wrapping the files etc.

We compiled and launched our graphical application with every possible optimization parameter. Our benchmark analysis includes CPU usage, memory usage, rendering speed (FPS), size of the resulting code, and speed of the translation process. The study was extended to the original native C/C++ application as well. The measurements were made with 1,500 new particles/frame. The results are summarized in the following table.

Table 6: Benchmarks

| **optimization level** | **CPU usage [%]** | **memory usage [MB]** | **speed [FPS]** | **size of the JS code [KB]** | **translation speed [sec]** |
|---|---|---|---|---|---|
| original C/C++ | $\sim 23.7$ | $\sim 84.9$ | $\sim 53$ | 429 | 0.33 |
| O0 (none) | $\sim 12.1$ | $\sim 35$ | $\sim 115$ | 2686 | 5.07 |
| O1 (js) | $\sim 9.4$ | $\sim 31$ | $\sim 142$ | 2169 | 4.81 |
| O2 (js) | $\sim 8.2$ | $\sim 26$ | $\sim 162$ | 581 | 6.62 |
| O3 (js) | $\sim 7.7$ | $\sim 19$ | $\sim 162$ | 553 | 6.79 |
| Os (LLVM) | $\sim 8.1$ | $\sim 27$ | $\sim 162$ | 546 | 6.80 |
| Oz (LLVM) | $\sim 8.0$ | $\sim 28$ | $\sim 160$ | 549 | 6.83 |

## 6. Summary

With the help of the Emscripten technology, application codes written in almost any high level languages (such as C/C++, Python etc.) can be transformed into JavaScript code, thus the original applications can be used on the web, or on even mobile devices. This technology can optimize source codes on many ways during the translation process. This feature is available due to the LLVM (Low Level Virtual Machine) compiler framework. Emscripten is accessible through a command-line environment and if the user would like to use various technologies (SDL, GLFW, OpenGL 1.x methods etc.), load files (models, textures, fonts etc.) or select optimization level and several different other options are available.

A C/C++ graphical application has been implemented for the purpose of doing benchmark tests. The JS code can run in a browser, in WebGL environment. The required HTML file is created by the Emscripten too. We used OpenGL 3.3, GLSL, GLFW3, and GLM libraries for the graphical visualization. Our measurements 2include CPU utilization, and memory usage, FPS (frames per second) values, size of the codes and translation speed. The tests were performed with all the available optimization techniques. The source code of sample application is available here: http://goo.gl/YmV2EA.

This paper contains the theoretical foundations for the technology acquisition, including JavaScript, LLVM and Emscripten. We have presented practical usage of the workspace and our experiences through our sample application.

**Acknowledgements**

The described article was carried out as part of the EFOP-3.6.1-16-2016-00011 Younger and Renewing University Innovative Knowledge City institutional development of the University of Miskolc aiming at intelligent specialisation project implemented in the framework of the Szechenyi 2020 program. The realization of this project is supported by the European Union, co-financed by the European Social Fund.

## REFERENCES

[1] TIOBE Index for February 2016, `http://www.tiobe.com/tiobe_index?page=index`

[2] D. Flanagan. *JavaScript: The Definitive Guide.* O'Reilly Media, 2006

[3] C. Prabhakar. *Google Web Toolkit: GWT Java Ajax Programming.* Packt Publishing, 2007

[4] F. Loitsch and M. Serrano. Hop *Client-Side Compilation. InTrends in Functional Programming,* SetonHall University, Intellect Bristol, 2008

[5] T. Petek and D. Syme. *AFAX: Rich client/server web applications in F#.* Draft. Retrieved April 2011

[6] `http://www.llvm.org`

[7] C. Lattner and V. Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,* 2004

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. *Efficiently computing static single assignment form and the control dependence graph.* October 1991

[9] Chris Lattner. *LLVM,* 2004.

[10] Official homepage of Emscripten: `http://www.emscripten.org`

[11] Alon Zakai. *Emscripten: An LLVM-toJava-script Compiler,* 2013

[12] Joel Galenson, Cindy Rubio-González, Sarah Chasins, Liang Gong. *Research.js: Evaluating Research Tool Usability on he Web,* University of California, Berkeley, 2014

[13] Malek Musleh, Vijay S. Pai. *Architectural Characterization of Client-side JavaScript Workloads & Analysis of Software Optimizations,* Purdue University, 2015

[14] Source code of Regal tool: `https://github.com/p3/regal`